Out-of-Core FFTs with Parallel Disks

Thomas H. Cormen* David M. Nicol†
Dartmouth College Department of Computer Science
Hanover, NH 03755
{thc,nicol}@cs.dartmouth.edu

Abstract

We examine approaches to computing the Fast Fourier Transform (FFT) when the data size exceeds the size of main memory. Analytical and experimental evidence shows that relying on native virtual memory with demand paging can yield extremely poor performance. We then present approaches based on minimizing I/O costs with the Parallel Disk Model (PDM). Each of these approaches explicitly plans and performs disk accesses so as to minimize their number.

1 Introduction

Although in most cases, Fast Fourier Transforms (FFTs) can be computed entirely in the main memory of a computer, in a few exceptional cases, the input vector is too large to fit. One application that uses very large FFTs is seismic analysis [2]; in one industrial application, an out-of-core one-dimensional FFT is necessary (as part of a higher dimensional FFT) even when the computer memory has 16 gigabytes of available RAM. Another application is in the area of radio astronomy. The High-Speed Data Acquisition and Very Large FFTs Project at Caltech[1] uses FFTs to support searching for fast (millisecond period) pulsars. The project currently requires FFTs with 10 gigapoints, and it desires FFTs with up to 64 gigapoints. One must use out-of-core FFT methods in such cases.

In out-of-core methods, data are stored on disk and repeatedly brought into memory a section at a time, operated on there, and written out to disk. Because disk accesses are so much slower than main memory accesses (typically at least 10,000 times slower), efficient out-of-core methods focus on reducing disk I/O costs. We can reduce disk I/O costs in two ways: reduce the cost of each access, and reduce the number of accesses.

We can reduce the per-access cost by using parallel disk systems. That is, we take advantage of the increase in I/O bandwidth provided by using multiple disks. If we use D disks instead of one disk, the I/O bandwidth may increase by up to a factor of D. Parallel disk systems are available on most parallel computers, and they are relatively simple to construct on networks of workstations.

In this paper, we shall concentrate on reducing the number of parallel disk accesses for performing out-of-core FFTs; Section 2 presents background material on FFTs. Section 3 shows how poorly they perform when they run with demand paging. Section 4 describes the Parallel Disk Model (PDM) of Vitter and Shriver [16], which we use to measure I/O costs for our explicit out-of-core algorithms.

This paper surveys approaches from two of our earlier papers:

1. In [6], we modify the Cooley-Tukey method to perform FFTs on a uniprocessor with parallel disks. Section 5 will review the method of that paper and show that it is a significant improvement over using demand paging.

2. In [8], we extend the uniprocessor method to multiple processors. Section 6 will discuss the four variations of a multiprocessor method that we implemented. In two of these variations, no communication occurs during butterfly computations, even though we are running on a multiprocessor. Instead, communication is folded into permutation operations that are performed anyway.

The methods described in Sections 5 and 6 substantially restructure the FFT computation by introducing permutation steps to maintain data locality. Consequently, they are far beyond anything we can expect a compiler or custom paging policies to do, as in the approaches of [10, 13].

2 FFTs

This section reviews Fourier transforms and outlines some well-known FFT methods for in-core computation. For further background on the FFT, see any of the texts [5, 12, 15].

The FFT is a particular method of computing the *Discrete Fourier Transform (DFT)* of an N-element vector. Given a vector $a = (a_0, a_1, \ldots, a_{N-1})$, where N is a power of 2, the *Discrete Fourier Transform (DFT)* is a vector $y = (y_0, y_1, \ldots, y_{N-1})$ for which

$$y_k = \sum_{j=0}^{N-1} a_j \omega_N^{jk} \qquad \text{for } k = 0, 1, \ldots, N-1 \ , \qquad (1)$$

*Supported in part by funds from Dartmouth College and in part by the National Science Foundation under grants CCR-9308667 and CCR-9625894.

†Supported in part by NSF grants CCR-9201195 and NCR-9527163, and supported in part by NASA Contract NAS1-19480 to the Institute for Computer Applications in Science and Engineering.

Portions of this article are reprinted from *Parallel Computing*, Thomas H. Cormen and David M. Nicol, "Performing out-of-core FFTs on parallel disk systems," to appear, with kind permission from Elsevier Science - NL, Sara Burgerhartstraat 25, 1055 KV Amsterdam, The Netherlands.

[1] See http://www.cacr.caltech.edu/SIO/APPL/phy02.html.

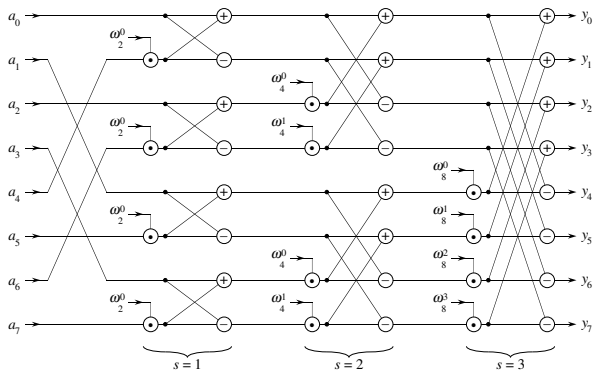


Figure 1: The FFT computation after fully unrolling the recursion, shown here with $N = 8$. Inputs $(a_0, a_1, \ldots, a_{N-1})$ enter from the left and first undergo a bit-reversal permutation. Then $\lg N = 3$ stages of butterfly operations are performed, and the results $(y_0, y_1, \ldots, y_{N-1})$ emerge from the right. This figure is taken from [5, p. 796].

where $\omega_N = e^{2\pi i/N}$ and $i = \sqrt{-1}$. For any real number u, we can directly compute $e^{iu} = \cos(u) + i\sin(u)$.

Figure 1 shows the butterfly graph as it is used in the *Cooley-Tukey method* of computing an FFT, drawn for $N = 8$. First, the input vector undergoes a bit-reversal permutation. A *bit-reversal permutation* is a bijection in which the element whose index k in binary is $(k_{N-1}, k_{N-2}, \ldots, k_0)$ maps to the element whose index in binary is $(k_0, k_1, \ldots, k_{N-1})$. After the bit-reversal permutation, a butterfly graph of $\lg N$ stages is computed. In the sth stage of the butterfly graph, elements whose indices are 2^s apart (after the bit-reversal permutation) participate in a butterfly operation. The butterfly operations in the sth stage can be organized into $N/2^s$ *groups* of 2^s operations each. Each butterfly operation has a third input, known as a *twiddle factor*. The twiddle factor for a butterfly operation in stage s and the jth butterfly within a group $(0 \le j < 2^{s-1})$ is $\omega_{2^s}^j$. The sequential running time is $\Theta(N \lg N)$.

Several other methods have been developed to improve performance on vector machines and in memory hierarchies, by avoiding the bit-reversal permutation to improve locality of reference. *Stockham's method* [15, pp. 49–58] eliminates bit-reversal by incorporating permutations into each of the $\lg N$ stages of the butterfly network. Its memory requirement, however, is twice that of the Cooley-Tukey method.

Swarztrauber's method uses a divide-and-conquer approach by splitting the summation of equation (1) into $\sqrt{N}$ summations each with $\sqrt{N}$ terms. The DFT in each subproblem is comprised of all terms whose indices are congruent modulo $\sqrt{N}$. The analog of a butterfly operation adds $\sqrt{N}$ terms—also expressible as DFTs—that are computed by recursive calls to subproblems of size $\sqrt{N}$. When N is small enough that the original problem fits in core, subproblems of size $\sqrt{N}$ fit in cache on most machines, and so Cooley-Tukey is then a good way to solve them.

3 Performance of FFTs with demand paging

In this section, we show that the in-core FFT methods described earlier perform poorly under demand paging once the problem size exceeds the available memory. The number of page faults for the Cooley-Tukey bit-reversal computation is at least $N/4$, and even under the best of conditions the butterfly steps for all methods suffer from a poor computation-to-I/O ratio. We substantiate our conclusions with experimental results.

Analysis of bit reversal

The following pseudocode expresses an in-place bit-reversal permutation of N-element array A:

```
for j ← 0 to N − 1
    do let j′ be the lg N-bit reversal of j
       if j < j′
          then exchange A[j] ↔ A[j′]
```

In [6], we prove the following theorem, which shows that the number of page faults for the Cooley-Tukey bit-reversal computation is at least $N/4$.

Theorem 1 *Suppose that the in-place bit-reversal permutation code above is performed under demand paging with least-recently-used page replacement. Suppose further that there are $N = 2^n$ elements in the array, the physical memory can hold $M = 2^m$ elements, and each page holds $B = 2^b$ elements, where n, m, and b are positive integers, $N \ge 2M$, and $N \ge 2B$. Finally, assume that the array A starts at a page boundary and that no pages of A are initially in memory. Then the bit-reversal permutation induces at least $N/4$ page faults.* ■

Analysis of butterfly stages

All of the FFT methods that we have discussed exhibit relatively good locality when executing each butterfly stage. For both Cooley-Tukey and Stockham, each butterfly stage essentially sweeps through all the data pages, exactly once, with no more than 2 data pages actively in use at a time. Swarztrauber's method exhibits more complex behavior, but its constituent butterflies act like the other two methods. The essential point to be noted is that during a butterfly stage, each data point is updated once by a complex addition/subtraction (two floating-point operations), and half the data points also involve a complex multiplication (six floating-point operations). A typical 8 KB data page contains 512 points, and so it entails 2560 floating-point operations. The time required to fault in a data page is on the order of 10^{-2} seconds (most of which is

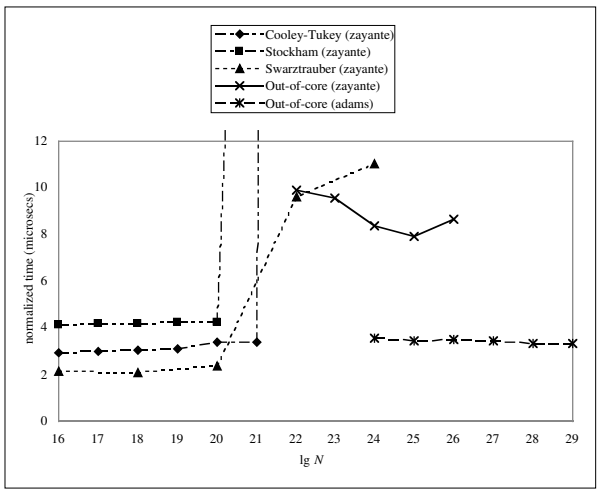


Figure 2: Normalized times in microseconds for uniprocessor FFT methods on a workstation (zayante) and a DEC 2100 server (adams) as a function of the problem size N in points (multiply by 16 for the problem size in bytes). Normalized time is the time per butterfly operation, or total time divided by $(N/2)\lg N$. Normalized times for the Cooley-Tukey and Stockham methods are too high to show. For Cooley-Tukey, with $N = 2^{21}$ the normalized time is 251.24 microseconds, and with $N = 2^{22}$ the normalized time is 441.12 microseconds. For Stockham's method, normalized times for $N = 2^{21}$ though $N = 2^{24}$ range from 42.48 to 44.14 microseconds. The out-of-core method timings use asynchronous I/O.

independent of the page size), but the time to process that page is about an order of magnitude less. Even with much better locality than the bit-reversal computation, demand-paged FFT suffers greatly from waiting for I/O to complete. We can mitigate this bottleneck by either increasing the size of block fetched per I/O, and/or by prefetching memory blocks. Our out-of-core techniques do both.

Experimental results

Here we present running times of the three demand-paged in-core FFT methods (Cooley-Tukey, Stockham, and Swarztrauber). They were coded in C, compiled using gcc with O2 optimization, and run on a DEC 3000 Alpha-based workstation running Digital UNIX V3.2C. The workstation, named zayante, has a clock cycle of 175 MHz, 64 MB of memory, and a 512 MB virtual-address space.

Figure 2 shows normalized running times, or the time spent per butterfly operation. The Cooley-Tukey and Swarztrauber methods both use $16N$ bytes; Stockham uses $32N$ and so experiences heavy paging one problem size earlier than the others. Because our implementation of Swarztrauber's method requires N to be a power of 4, timings for odd powers of 2 are omitted.

From Figure 2, we see the effects of demand paging. By avoiding bit-reversal, the Stockham and Swarztrauber methods do not experience the degree of thrashing suffered by Cooley-Tukey. (In fact, we did not even run Cooley-Tukey for $N = 2^{24}$, anticipating a run time of about a day.) Swarztrauber's method is notably faster in each case, probably due to its substantially better locality in cache. Nevertheless, we shall see in Section 5 (and Figure 2 shows) that our explicit out-of-core algorithm runs faster than even Swarztrauber's method on the same system for a problem size of $N = 2^{24}$.

4 The Parallel Disk Model

This section describes the Parallel Disk Model [16], which underlies all the out-of-core FFT algorithms herein.

In the *Parallel Disk Model*, or *PDM*, N records are stored on D *disks* $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, with N/D *records* stored on each disk. For our purposes, a record is a complex number comprised of two 8-byte double-precision floats. The records on each disk are partitioned into *blocks* of B records each. Any disk access transfers an entire block of records. Disk I/O transfers records between the disks and an M-record *random-access memory*. Any set of M records is a *memoryload*.

We assess an algorithm by the number of parallel I/O operations it requires. Each *parallel I/O operation* transfers up to D blocks between the disks and memory, with at most one block transferred per disk, for a total of up to BD records transferred. The most general type of parallel I/O operation is *independent I/O*, in which the blocks accessed in a single parallel I/O may be at any locations on their respective disks. A more restricted operation is *striped I/O*, in which the blocks accessed in a given operation must be at the same location on each disk.

For the multiprocessor algorithm, we assume that there are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ connected by a network. Network speeds vary greatly, but for the multiprocessors that we consider, interprocessor communication times are far less than I/O latencies. The M-record memory is distributed among the P processors so that each processor holds M/P records. The implementation of the PDM we use is the ViC* API [4], in which $D \geq P$ and each processor $\mathcal{P}_i$ communicates only with the D/P disks $\mathcal{D}_{iD/P}, \mathcal{D}_{iD/P+1}, \ldots, \mathcal{D}_{(i+1)D/P-1}$. (If $D < P$ in a given physical configuration, the ViC* implementation provides the illusion that $D = P$ by sharing each physical disk among P/D processors.)

We place some restrictions on the PDM parameters. We assume that P, B, D, M, and N are exact powers of 2. For convenience, we define $p = \lg P$, $b = \lg B$, $d = \lg D$, $m = \lg M$, and $n = \lg N$. We assume that $BD \leq M$ so that the memory can hold the contents of one block from each disk, and of course we assume that $M < N$ so that the problem is out-of-core.

The PDM lays out data on a parallel disk system as shown in Figure 3. A *stripe* consists of the D blocks at the same location on all D disks. A record's index is an n-bit vector. In Section 6, we will take advantage

	$\mathcal{P}_0$		$\mathcal{P}_1$		$\mathcal{P}_2$		$\mathcal{P}_3$	
	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{D}_7$
stripe 0	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
stripe 1	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31
stripe 2	32 33	34 35	36 37	38 39	40 41	42 43	44 45	46 47
stripe 3	48 49	50 51	52 53	54 55	56 57	58 59	60 61	62 63

Figure 3: The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

of interpreting a record index as a sequence of bit fields that give the record's location in the parallel disk system; from most significant bits to least significant bits, the bit fields are

- $\lg(N/BD) = n-(b+d)$ bits containing the number of the stripe (since each stripe has BD records, there are N/BD stripes),
- $\lg D = d$ bits containing the disk number; of these, the most significant $\lg P = p$ contain the processor number,
- $\lg B = b$ bits containing the record's offset within its block.

Since each parallel I/O operation accesses at most BD records, any algorithm that must access all N records requires $\Omega(N/BD)$ parallel I/Os, and so $O(N/BD)$ parallel I/Os is the analogue of linear time in sequential computing. Vitter and Shriver showed a tight bound of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os for the FFT, which appears to be the analogue of the $\Theta(N \lg N)$ bound seen for so many sequential algorithms on the standard RAM model.

5 Uniprocessor method

By designing explicitly for the PDM, we can get considerably better out-of-core FFT performance than we get by using just demand paging. The key idea is to redraw the butterfly graph by inserting permutations. We then recognize that bit-reversal and the added permutations belong to the larger class of BMMC permutations. We use a prior out-of-core BMMC algorithm to produce an efficient out-of-core FFT.

BMMC permutations on the PDM

A *BMMC* (bit-matrix-multiply/complement) permutation on $N = 2^n$ elements is specified by an $n \times n$ *characteristic matrix* $H = (h_{ij})$ whose entries are drawn from $\{0,1\}$ and is nonsingular (i.e., invertible) over $GF(2)$.[2] Treating each source index x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ to form the corresponding n-bit target index z: $z = H\,x$. As long as the characteristic matrix H is nonsingular, the mapping of source indices to target indices is one-to-one.

An efficient algorithm for BMMC permutations on the PDM appears in [7]. This algorithm requires at most $\frac{2N}{BD}\left(\left\lceil\frac{\operatorname{rank}\phi}{\lg(M/B)}\right\rceil + 1\right)$ parallel I/Os, where ϕ is the lower left $\lg(N/M) \times \lg M$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. (Note that because of the dimensions of ϕ, its rank is at most $\lg\min(M, N/M)$.) This number of factors is asymptotically optimal and is close to the best known exact lower bound.

We shall use two types of BMMC permutations to perform the FFT.

Bit-reversal permutation: The characteristic matrix has 1s on the antidiagonal and 0s elsewhere. The submatrix ϕ has as much rank as possible, so that $\operatorname{rank}\phi = \lg\min(M, N/M)$.

k-bit right-rotation: We rotate the bits of each index k bits to the right, wrapping around at the rightmost position. The characteristic matrix is formed by taking the identity matrix and rotating its columns k positions to the right, and $\operatorname{rank}\phi \le \min(k, \lg N - k, \lg M, \lg(N/M))$.

Redrawing the butterfly

Figure 4 shows the structure of our algorithm. This redrawing of the butterfly was devised by Snir [14] and is implicitly used in the FFT algorithm of Vitter and Shriver [16]. We use an *effective memory size* F. Assume for now that $F = M$ and that $\lg F$ divides $\lg N$. As in the Cooley-Tukey method, we start with a bit-reversal permutation. Then there are $\lg N/\lg F$ *superlevels*, where each superlevel consists of N/F separate

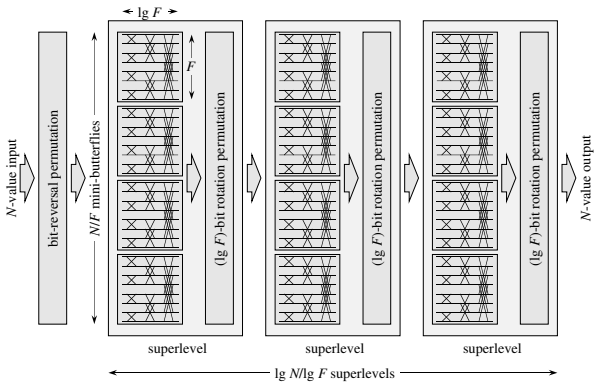


Figure 4: The structure of the out-of-core FFT algorithm for the PDM. After a bit-reversal permutation, we perform $\lg N/\lg F$ superlevels. Each superlevel consists of N/F mini-butterflies on F values, followed by a $(\lg F)$-bit right-rotation permutation on the entire array.

[2] Matrix multiplication over $GF(2)$ is like standard matrix multiplication over the reals but with all arithmetic performed modulo 2. Equivalently, multiplication is replaced by logical-and, and addition is replaced by exclusive-or. Technically, the specification of a BMMC permutation also includes a "complement vector" of length n, but we will not need complement vectors in this paper.

"mini-butterflies" followed by a $(\lg F)$-bit right-rotation permutation on the entire array.

Each *mini-butterfly* is a butterfly graph on F values, and hence it has depth $\lg F$ and a sequential running time of $\Theta(F \lg F)$. The size F of a mini-butterfly is chosen so that each mini-butterfly is computed by reading in a memoryload, computing the mini-butterfly graph, and writing out the memoryload.

Analysis

This FFT algorithm consists of one bit-reversal permutation followed by $\lg N / \lg F$ superlevels. As noted above, the bit-reversal permutation requires at most $\frac{2N}{BD}\left(\left\lceil \frac{\lg \min(M, N/M)}{\lg(M/B)} \right\rceil + 1\right)$ parallel I/Os. Each superlevel requires $2N/BD$ parallel I/Os to read and write all N/F mini-butterflies plus at most $\frac{2N}{BD}\left(\left\lceil \frac{\lg \min(F, N/F, M, N/M)}{\lg(M/B)} \right\rceil + 1\right)$ parallel I/Os to perform the $(\lg F)$-bit rotation permutation. Since we assume that $F = M$, the $\lg F$ factor in the numerator drops out. Asymptotically, the number of parallel I/O operations is $\Theta\left(\frac{N}{BD} \frac{\lg N}{\lg M} \left(1 + \frac{\lg \min(M, N/M)}{\lg(M/B)}\right)\right)$, which can be shown via simple manipulations to equal the lower bound of $\Omega\left(\frac{N}{BD} \frac{\lg(N/B)}{\lg(M/B)}\right)$ proven by Aggarwal and Vitter [1].

Implementation notes

We briefly mention some implementation details; see [6] for more information.

- If $\lg F$ does not divide $\lg N$, then we compensate in the last superlevel, computing mini-butterflies of depth $(\lg N) \bmod (\lg F)$.
- The BMMC permutations affect the twiddle-factor computations. We can still compute twiddle factors efficiently.
- The BMMC permutation subroutine is taken from the implementation in [4]. It calls the ViC* API to perform striped reads and independent writes. It is carefully optimized for both in-core computation and I/O.
- We implemented the FFT algorithm with both synchronous (i.e., blocking) and asynchronous (non-blocking) I/O calls; the ViC* interface supports both.

Performance

This section concludes with timing results for the out-of-core FFT algorithm on two different DEC Alpha platforms. In all cases, block sizes were 2^{16} bytes.

We start with a direct comparison of our algorithm and the in-core methods running with demand paging on zayante. Figure 2 shows the normalized running times. With our algorithm, we used $D = 1$ disk, asynchronous I/O, and a memory size of 2^{25} bytes, or 2^{21} records. Using only one disk for the our algorithm makes for a fair comparison to demand paging, since there is only one swap disk. At the problem sizes at which the in-core algorithms encounter heavy paging—$N \geq 2^{22}$ for Cooley-Tukey and Swarztrauber—our out-of-core algorithm is faster. (At $N = 2^{23}$, our algorithm is over 46 times faster than Cooley-Tukey.) Considering the overhead due to the ViC* wrappers and UFS calls, it is impressive that our algorithm can run faster than even Swarztrauber's method.

Figure 2 also shows the normalized running time on a different system, named adams, which is a DEC 2100 server with two 175-MHz Alpha processors, 320 MB of memory, and eight 2-GB disks for data (so that $D = 8$). Of the 320 MB of memory, the out-of-core method uses 2^{27} bytes, or 128 MB. It has the same software environment as zayante, but with eight disks, its I/O bandwidth is much higher. The operating system may choose to run a ready thread on either processor, and so disk-server threads do not interfere with butterfly computations as much as on zayante. Compared to the in-core methods in Figure 2 even when they run entirely in memory, the normalized times (which do include I/O time) are at worst slightly higher and in some cases even lower! In one case not shown in the figure ($N = 2^{23}$, 2^{26} bytes of memory used, and synchronous I/O), the running time on adams is 144.7 times lower than Cooley-Tukey on zayante.

6 Multiprocessor methods

In this section, we describe the modifications to the uniprocessor out-of-core FFT algorithm that enable it to work on multiple processors. We start by looking at a straightforward way to extend the uniprocessor algorithm to use multiple processors. Because each mini-butterfly spans all P processors and the data layout is stripe by stripe, this method has nonuniform communication characteristics and is difficult to implement. By laying out the data differently (changing the band size) and keeping each mini-butterfly within a processor (changing the effective memory size), we derive an algorithm that has no interprocessor communication during the mini-butterfly computations.

The straightforward multiprocessor algorithm

Conceptually, the uniprocessor algorithm in Section 5 is already a multiprocessor algorithm if viewed in the right way. Suppose we choose the effective memory size F to be the memory size M over all P processors. (Assume for the moment that we use synchronous I/O so that reducing the effective memory size for additional buffers is not an issue.) Then we can compute each mini-butterfly by simply computing the butterfly graph (e.g., Figure 1) on P processors, subject to the twiddle factors being altered as mentioned in Section 5.

In reality, however, the multiprocessor algorithm is not quite so simple. Consider the data layout in Figure 3, and suppose that the effective memory size F is 2 stripes, or 32 records. Observe that the F/P records

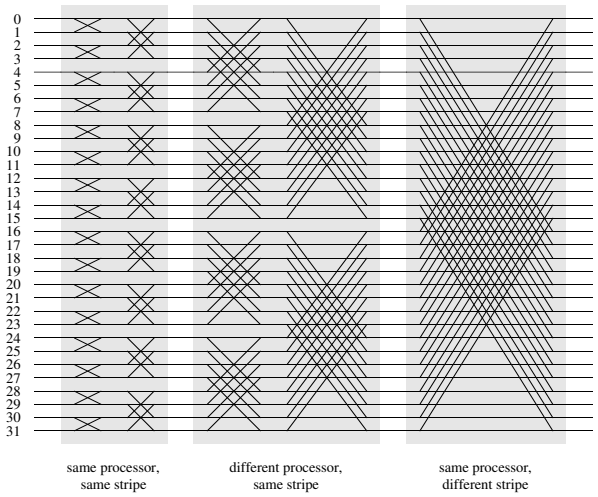


Figure 5: Computing a mini-butterfly when the band size is $\beta = BD$ and the effective memory size is $F > \beta/P$. Here, $B = 2$, $P = 4$, $D = 8$, and $F = 32 = 2BD$. Indices on the left are record numbers in the first mini-butterfly. Twiddle factors are omitted. The first $\lg(BD/P) = 2$ stages use computation internal to each processor, where each butterfly operation uses two values from the same stripe. The next $\lg P = 2$ stages require interprocessor communication to exchange values from the same stripe between processors. The last $\lg(F/BD) = 1$ stages use more computation internal to each processor, where each butterfly operation uses two values from different stripes.

that map to a given processor in a mini-butterfly are not all consecutive. Processor $\mathcal{P}_0$, for example, holds records 0 to 3 and 16 to 19 in the first mini-butterfly. Figure 5 shows what happens when we compute the first mini-butterfly in this situation. Each butterfly operation involves two records whose indices differ in exactly one bit. There are three different communication characteristics, depending on which bit differs.

1. Each processor has BD/P consecutive records from a given stripe. In the first $\lg(BD/P)$ stages, therefore, computation is internal to each processor.

2. Each butterfly operation in the next $\lg P$ stages involves two records from the same stripe but in different processors. Thus, each such operation requires interprocessor communication.

3. In the last $\lg(F/BD)$ stages, each butterfly operation involves two records that are from different stripes but are in the memory of the same processor. These $\lg(F/BD)$ stages, therefore, use only internal computation.

Because there are three different communication characteristics that depend on stage numbers, we found this

(a) $\beta = F$

$\mathcal{P}_0$				$\mathcal{P}_1$				$\mathcal{P}_2$				$\mathcal{P}_3$			
$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
0	1	2	3	8	9	10	11	16	17	18	19	24	25	26	27
4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31
32	33	34	35	40	41	42	43	48	49	50	51	56	57	58	59
36	37	38	39	44	45	46	47	52	53	54	55	60	61	62	63

(b) $\beta = N$

$\mathcal{P}_0$				$\mathcal{P}_1$				$\mathcal{P}_2$				$\mathcal{P}_3$			
$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63

Figure 6: Layouts with different band sizes in the same configuration as Figure 3. The effective memory size $F = 32$ is shown by double horizontal lines.

algorithm quite tricky to implement. Add in the twiddle factors, memory addressing (consider that in $\mathcal{P}_0$, records 0 to 3 and 16 to 19 are in eight consecutive memory locations), and changes for when $\lg F$ does not divide $\lg N$, and the code becomes rather long and difficult to get right. Even without the ViC* API calls for disk I/O, it is several pages long.

Band size

Part of the problem with the above approach is that the data layout and effective memory size do not interact well. We define the *band size* β of a data layout on a parallel disk system as the maximum number of consecutive records per processor times the number of processors. By "consecutive records per processor," we mean that the records should be consecutively indexed on the processor's disks. In Figure 3, each processor has 4 consecutive records on its disks, and so the band size β is $4P = 16$, which equals the stripe size $BD = 16$. In Figure 6(a), each processor has 8 consecutive records on its disks, and so the band size is $8P = 32$; since $F = 2BD = 32$, we have that $\beta = F = 2BD$. In Figure 6(b), each processor has 16 consecutive records on its disks, and so the band size is $16P = 64$, which equals the total number of records N.

When the portion of a band that resides in one processor (β/P records) is smaller than the effective memory size F, computing a mini-butterfly must induce interprocessor communication. Allowing for the band size to be even smaller—$\beta < F$—then we have the situation above in which there are three different communication characteristics.

Varying the effective memory size and band size

In fact, we can vary the effective memory size and band size to simplify the mini-butterfly computations.

So far, we have considered an effective memory size of $F = M$. Suppose instead that we use $F = M/P$, so that each mini-butterfly is the size of an individual processor's memory, and suppose further that we can change the data layout to have a band size $\beta \geq M$. Then we could compute each mini-butterfly without any communication at all, since each mini-butterfly would consist of M/P consecutive records within the same processor.

Changing the data layout turns out to be fairly simple. We change the bit-reversal and bit-rotation permutations performed in Figure 4 to other BMMC permutations. There are two important observations. First, conversions between power-of-2 band sizes are BMMC permutations; converting from one band size to another is actually a matter of "sliding" the $\lg P$ bits that give the processor number either left or right. Second, BMMC permutations are closed under composition.

Now we can see how to alter the BMMC permutations used in the FFT algorithm. The BMMC permutation subroutine assumes that the records are laid out on the disks with a band size of BD, but the reading and writing of mini-butterflies assumes a band size of some value β. Suppose that the $n \times n$ matrix T characterizes a $(\lg F)$-bit right rotation permutation with band size BD. Let the $n \times n$ matrix Π characterize the BMMC permutation that converts a band size of β to a band size of BD. Let Π^{-1} be the inverse of Π, so that Π^{-1} characterizes the BMMC permutation that converts a band size of BD to a band size of β. Then instead of just performing the permutation characterized by T, we first convert from band size β to band size BD, we then perform the permutation characterized by T, and finally we convert from band size BD back to band size β. In other words, we perform just one BMMC permutation, and it is characterized by the matrix product $(\Pi^{-1}T\Pi)$.

The above alteration works for all but the first and last BMMC permutations in the FFT algorithm. The first BMMC permutation differs in two ways: the records start out with band size BD rather than β, and it is a bit-reversal permutation. If the matrix R characterizes the bit-reversal permutation, then we perform the BMMC permutation characterized by the matrix product $(\Pi^{-1}R)$. The last BMMC permutation also differs in two ways: it may be a bit rotation by fewer than $\lg F$ bits, and the records end up with band size BD rather than β. If the matrix T' characterizes the bit-rotation permutation in the last superlevel, then we perform the BMMC permutation characterized by the matrix product $(T'\Pi)$.

Meaningful combinations of effective memory size and band size

Later, we present performance results for two effective memory sizes (M and M/P) and three band sizes (BD, M, and N). Of these six combinations, only four make sense to implement. The two that do not are $F = M/P$, $\beta = BD$ and $F = M$, $\beta = N$. In both cases, each mini-butterfly would not contain F consecutive records, and the computation would not map well into the structure shown in Figure 4.

Recalling the interpretation of bit fields in a banded layout, we name the banded layouts we use as follows:

Stripe-major: $\beta = BD$. Each band is a stripe, so that the most significant $\lg(N/BD)$ bits give both the band number and the stripe number.

Memoryload-major: $\beta = M$. Each band is a memoryload, so that the most significant $\lg(N/M)$ bits give both the band number and the memoryload number.

Processor-major: $\beta = N$. There is only one band, and it consists of all N points. Since $\lg(N/\beta) = 0$ in this case, the most significant $\lg P$ bits give the processor number.

Of the four meaningful combinations, the two with $F = M/P$ require no interprocessor communication during the mini-butterfly computations. We call these the *no-communication methods*, in contrast to the *communication methods*, in which $F = M$. It is remarkably simple to modify the uniprocessor FFT code to implement the no-communication methods.

We name the four methods examined as follows:

1. Stripe-major/communication: $\beta = BD$, $F = M$.
2. Memoryload-major/communication: $\beta = M$, $F = M$.
3. Processor-major/no-communication: $\beta = N$, $F = M/P$.
4. Memoryload-major/no-communication: $\beta = M$, $F = M/P$.

Effect on I/O complexity

One can show that the I/O complexity is $O\left(\frac{N}{BD}\frac{\lg N}{\lg F}\left(1 + \frac{\lg \min(B, N/B)}{\lg(M/B)}\right)\right)$. Asymptotically, varying the band size has no effect on the I/O complexity.

When $F = M$, the I/O complexity is optimal, since one can show that $O\left(\frac{N}{BD}\frac{\lg N}{\lg M}\left(1 + \frac{\lg \min(B, N/B)}{\lg(M/B)}\right)\right) = O\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$. Reducing the effective memory size F from M to M/P increases the asymptotic I/O complexity, for there may be additional superlevels which introduce additional I/O.

In practice, these additional superlevels occur rarely. Consider a configuration with 16 megabytes of memory per processor, which works out to $M/P = 2^{20}$ records per processor. Additional superlevels occur when there are many processors, so suppose that $P = 256$ and hence $M = 2^{28}$. The number of superlevels is $\lceil \lg N / \lg F \rceil$. When $F = M$, there are two superlevels for all N in the range 2^{29} to 2^{56}. When

$F = M/P$, the range of N for which there are two superlevels is smaller—2^{21} to 2^{40}—but it still includes the largest problem size we are likely to see for some time to come.

Effect on communication complexity

Analyzing the change in communication complexity with varying memory size and band size is difficult. Of course, when $F = M/P$, there is no communication when computing the mini-butterflies. When $F = M$, we can determine the number of MPI messages and total volume of data communicated for a particular set of parameters; this calculation is complicated by differences in the last superlevel. Communication analysis becomes difficult when the band size changes. Because the characteristic matrices given to the BMMC subroutine change with the band size, the communication patterns within the BMMC subroutine change as well. We do not know of a purely analytical way to determine the exact nature of this change.

There are two non-analytical ways to determine the effect of band size on communication complexity. One is to instrument the FFT implementations to measure the number of MPI messages and communication volume; the results later on use this method. The other way does not require the FFT code to actually run. Because the entire FFT algorithm is both deterministic and oblivious (i.e., its control flow does not depend on the values of the N points), if we are given an exact set of parameters N, M, B, D, P, F, and β, then we can calculate the number of MPI messages and total communication volume. Later, we describe an analysis program that performs this calculation for each of the four methods considered, with variants for both synchronous and asynchronous I/O.

The primary question

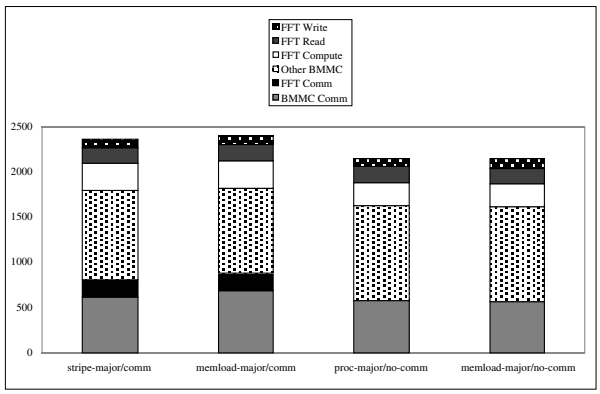
The primary question we ask is which effective memory size is better: M or M/P? That is, are the communication or no-communication methods faster? Under certain conditions, the no-communication methods may cause there to be more superlevels. And there may be a tradeoff in communication during mini-butterflies versus communication during BMMC permutations. With the no-communication methods, we can avoid all interprocessor communication during mini-butterfly computations, but the modified characteristic matrices may cause additional interprocessor communication during the BMMC permutations. Performance results and the analysis program we have written answer this question.

Performance of the multiprocessor methods

Here we present performance results for the out-of-core multiprocessor FFT methods. We shall see that when the number of superlevels is the same, the no-communication methods are faster.

The platform is "FLEET," a set of eight IBM RS6000 workstations connected by a 100-Mbit/second FDDI network. Each node runs AIX 4.1. Interprocessor communication is performed via the MPI calls `MPI_Sendrecv()` and `MPI_Sendrecv_replace()` using `mpich` version 1.1. Parallel I/O calls are through the ViC* API [4], which in turn makes calls to the Galley File System [11]. Galley uses separate I/O processes (IOPs) to manage parallel I/O calls. The ViC* API treats each IOP like a disk. On FLEET, it is fastest to run the IOPs on separate nodes from the computational processes. Consequently, we report results for $P = 4$ and $D = 4$. All runs were for $N = 2^{25}$ points (or 2^{29} bytes), which is the largest data set possible with this configuration of Galley on FLEET. We report on timing runs with synchronous I/O; the results for asynchronous I/O are essentially the same.

Figure 7 shows the results with synchronous I/O and a memory size of 2^{24} bytes per processor. There are two superlevels. The two no-



Method	Mini-butterfly messages	Mini-butterfly volume	BMMC messages	BMMC volume	Total messages	Total volume
stripe-major/ communication	64	268	480	503	544	771
memoryload-major/ communication	64	268	504	603	568	872
processor-major/no-communication	0	0	480	503	480	503
memoryload-major/no-communication	0	0	480	503	480	503

Figure 7: Results for one run of each of the four methods with synchronous I/O and 2^{24} bytes of memory per processor. The vertical axis is time, in seconds. From bottom to top, each stacked bar shows time spent in communication during the BMMC subroutine, communication time during mini-butterfly computations, non-communication time spent in the BMMC subroutine, computation time in mini-butterflies, time to read mini-butterflies, and time to write mini-butterflies. Below the chart is a table of how many messages and the volume of messages used by each of the four methods. Message measurements are per processor (all processors have the same communication characteristics), and all message volumes are to the nearest million bytes.

communication methods had virtually identical performance, and the two communication methods were very close as well. The total time is dominated by the BMMC subroutine for all four methods. The two no-communication methods are about 9.5% faster than the stripe-major/communication method.

The table included in Figure 7 shows the number of messages and message volume, rounded to the nearest million bytes, per processor for the mini-butterfly and BMMC portions of the computation. We see that in the communication methods, each processor sends 64 messages for a total of about 268 million bytes during the mini-butterfly computations and of course the no-communication methods send no messages during mini-butterfly computations. Three of the four methods have the same communication characteristics during the BMMC portion, but the memoryload-major/communication method sends more messages and more bytes.

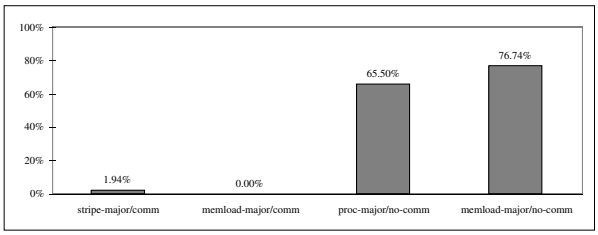


Figure 8: The result of running the analysis program for 258 problem instances with varying memory and problem sizes and both synchronous and asynchronous I/O. Percentages shown are how often each method sends the minimum number of messages *and* bytes over all four methods for a given instance and choice of synchronous/asynchronous I/O. Because there may be ties, and because a method may achieve the minimum in number of messages but not in number of bytes (or vice versa), the percentages shown do not add up to 100%.

Analysis program

The performance results above are for one problem size on one hardware/software configuration. Here we discuss a program that analyzes the communication requirements for all four methods.

The analysis program takes far less time than actual out-of-core FFT computations. It simply emulates the out-of-core FFT computation but omits I/O, butterfly computations, and most of the work in performing BMMC permutations. The program usually runs in under one second on a conventional workstation. Consequently, one can invoke the analysis program for each method at run time and choose the method with the lowest communication cost to actually run.

Given which of the four methods to use, and whether I/O is synchronous or asynchronous, the program produces four values: the number of messages in the BMMC permutation subroutine, the number of bytes sent in the BMMC subroutine, the total number of messages in the entire FFT computation, and the total number of bytes sent in the entire FFT computation.

Figure 8 shows the result of running the analysis program for 258 problem instances. Both memory size and problem size vary among these instances. Memory sizes range from 2^{17} bytes to 2^{35} bytes per processor, over four processors. Problem sizes range from 2^{20} points to 2^{35} points. For each combination of memory and problem size, both synchronous and asynchronous I/O are analyzed. The figure shows how often each method achieves the minimum over all four methods for both total number of messages *and* total number of bytes sent. For example, in 198 of the 258 instances (or 76.74%), the memoryload-major/no-communication method was at least as good as the other three methods in both number of messages and number of bytes sent. It is clear from Figure 8 that the two no-communication methods are usually the best.

The analysis program may be of help even if no method achieves the minimum in both messages and bytes for a given configuration and problem size. To a first-order approximation, communication time is usually a weighted sum of number of messages and number of bytes. Given the message and byte counts of each of the four methods, one can weight them appropriately and add them together to estimate communication time.

7 Conclusion

We have examined both analytically and experimentally classes of methods for computing large Fourier transforms. In-core FFT algorithms run slowly when they execute in a demand-paging environment. Of the three that we examined, Swarztrauber's method is by far the fastest and has the best locality of reference. The explicit out-of-core method that we developed for the PDM is asymptotically optimal in this model, and it has good empirical performance. On a DEC 2100 server with two processors, large memory, and eight data disks, our algorithm's normalized time is competitive with in-core methods, even when they run entirely in memory. We looked at four ways to perform out-of-core multiprocessor FFTs with distributed memory using the Parallel Disk Model. Overall, the best ways avoid interprocessor communication during the in-core mini-butterfly computations.

The advantage of the no-communication methods obviously depends on the relative speeds of processing and communication. Our initial multiprocessor implementation used an earlier version of `mpich`—version 1.0.13—which we found to be quite a bit slower than version 1.1. With communication costing more in version 1.0.13, the no-communication methods were about 14% faster than the communication methods. What are the future prospects of communication vs. computation? Without question, network speeds are improving. On the other hand, processor speeds are

improving and, even more important, programmers are becoming sensitive to keeping processors busy by using caches well. The performance gains from good cache management may well exceed network improvements, in which case the no-communication methods would gain even more of a relative advantage compared to the communication methods.

We alluded in Section 6 to one advantage of the no-communication methods: ease of developing code. Starting from a working out-of-core uniprocessor FFT program, it took *under an hour* of programming time to convert it to a multiprocessor program for the processor-major/no-communication method, and it worked the first time. In contrast, starting from the same point, it took *several weeks* to develop and debug the stripe-major/communication method.

Finally, we note that there are other frameworks for out-of-core FFTs with parallel disks. Rather than the Cooley-Tukey-based methods examined herein, [3] shows how to adapt Swarztrauber's method for an out-of-core setting on the PDM.

Acknowledgments

We thank Barry Fagin, Peter Highnam, David Keyes, and Jeff Rutledge for pointing us to applications of out-of-core FFTs, and also Dennis Healy and Eric Schwabe for their help in describing the mathematical structure of FFTs. Melissa Hirschl wrote the ViC* wrappers. David Kotz, Sanjay Khanna, and Wayne Cripps advised us in sundry systems issues. The DEC 2100 server named adams was funded in part by an equipment allowance from Digital Equipment Corporation. The FLEET Laboratory was funded by NASA Ames NAS through Agreement Number NAG 2-936 and by IBM through Dartmouth's IBM Matching Funds program.

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] Jon F. Claerbout. *Imaging the Earth's Interior*. Blackwell Scientific Publications, 1985.

[3] Thomas H. Cormen. Determining an out-of-core FFT decomposition strategy for parallel disks by dynamic programming. Technical Report PCS-TR97-322, Dartmouth College Department of Computer Science, July 1997. To appear in [9].

[4] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[6] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. Technical Report PCS-TR96-294, Dartmouth College Department of Computer Science, August 1996. To appear in *Parallel Computing*.

[7] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, July 1994. Preliminary version appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*. Revised version to appear in *SIAM Journal on Computing*.

[8] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 68–78, November 1997.

[9] *Proceedings of the Workshop on Algorithms for Parallel Machines*, 1996–97 Special Year on Mathematics of High Performance Computing, Institute for Mathematics and Its Applications, University of Minnesota, Minneapolis, September 1996.

[10] Todd C. Mowry, Angela K. Demke, and Orran Kreiger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 3–17, October 1996.

[11] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[12] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, New York, second edition, 1982.

[13] Yoonho Park, Ridgway Scott, and Stuart Sechrest. Virtual memory versus file interfaces for large, memory-intensive scientific applications. In *Proceedings of Supercomputing '96*, November 1996.

[14] M. Snir. I/O limitations on multi-chip VLSI systems. In *Proceedings of the 19th Allerton Conference on Communication, Control and Computation*, pages 224–233, 1981.

[15] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, 1992.

[16] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.